FreeBSD® JOURNAL

March/April 2020

File Systems

OpenZFS Encryption Arrives on FreeBSD

FUSE Driver Update

The Humble Bridge

TrueNAS® M-SERIES
Powerfully Scalable Enterprise Storage

OPEN STORAGE FOR ENTERPRISE WORKLOADS

UTILIZES FLASH-OPTIMIZED ZFS TECHNOLOGY

IDEAL FOR LATENCY-SENSITIVE AND BUSINESS-CRITICAL VIRTUAL MACHINES AND PHYSICAL WORKLOADS.

- PERFORMANCE AND SCALE WITHOUT COMPROMISE
- INTELLIGENT STORAGE OPTIMIZATION
- SELF-HEALING DATA PROTECTION
- UNLIMITED SNAPSHOTS AND REPLICATION

Contact iXsystems to Learn More about what TrueNAS® can do for your business!

iXsystems.com/TrueNAS | (855) GREP-4-iX

iXsystems™

© 2020 iXsystems. TrueNAS is a registered trademark of iXsystems, Inc. All rights reserved. Intel, the Intel logo, Xeon, and Xeon Inside are trademarks of Intel Corporation or its subsidiaries in the U.S. and/or other countries. All product names, logos, and brands are property of their respective owners.

FreeBSD® JOURNAL
Editorial Board

John Baldwin • FreeBSD Developer, Member of the FreeBSD Core Team and Chair of *FreeBSD Journal* Editorial Board.

Bryan Drewery • Senior Software Engineer at Dell EMC Isilon, member of FreeBSD Portmgr Team, and FreeBSD Committer.

Justin Gibbs • Founder of the FreeBSD Foundation, President of the FreeBSD Foundation, and a Software Engineer at Facebook.

Daichi Goto • Director at BSD Consulting Inc. (Tokyo).

Joseph Kong • Software Development Engineer at Amazon Web Services, author of *FreeBSD Device Drivers* and *Designing BSD Rootkits.*

Dru Lavigne • Author of *BSD Hacks* and *The Best of FreeBSD Basics.*

Michael W Lucas • Author of *Absolute FreeBSD.*

Ed Maste • Director of Project Development, FreeBSD Foundation.

Kirk McKusick • Treasurer of the FreeBSD Foundation Board, and lead author of *The Design and Implementation* book series.

George V. Neville-Neil • Director of the FreeBSD Foundation Board, and co-author of *The Design and Implementation of the FreeBSD Operating System.*

Philip Paeps • Secretary of the FreeBSD Foundation Board, FreeBSD Committer, and Independent Consultant.

Kristof Provost • Treasurer of the EuroBSDCon Foundation, FreeBSD Committer, and Independent Consultant.

Hiroki Sato • Director of the FreeBSD Foundation Board, Chair of Asia BSDCon, member of the FreeBSD Core Team, and Assistant Professor at Tokyo Institute of Technology.

Benedict Reuschling • Vice President of the FreeBSD Foundation Board, a FreeBSD Documentation Committer, and member of the FreeBSD Core Team.

Robert N. M. Watson • Director of the FreeBSD Foundation Board, Founder of the TrustedBSD Project, and University Senior Lecturer at the University of Cambridge.

S&W PUBLISHING LLC
PO BOX 408, BELFAST, MAINE 04915

Publisher • Walter Andrzejewski
walter@freebsdjournal.com

Editor-at-Large • James Maurer
jmaurer@freebsdjournal.com

Copy Editor • Annaliese Jakimides

Design & Production • Reuter & Associates

Advertising Sales • Walter Andrzejewski
walter@freebsdjournal.com
Call 888/290-9469

FreeBSD Journal (ISBN: 978-0-615-88479-0) is published 6 times a year (January/February, March/April, May/June, July/August, September/October, November/December).
Published by the FreeBSD Foundation,
3980 Broadway St. STE #103-107, Boulder, CO 80304
ph: 720/207-5142 • fax: 720/222-2350
email: info@freebsdfoundation.org
Copyright © 2020 by FreeBSD Foundation. All rights reserved. This magazine may not be reproduced in whole or in part without written permission from the publisher.

LETTER from the Board

Thank You to the Community

Welcome to the March/April issue of the *FreeBSD Journal*. On behalf of the Foundation, I wanted to take a minute to thank everyone for continuing to support FreeBSD during this challenging time. When recruiting folks to join the Project, we always talk about the community and how kind, generous, and welcoming it can be. This crisis has shown me that truer words were never spoken. From supporting other developers in need, to creating new methods of engagement to fill the void left by the end of in-person events, the FreeBSD community has gone above and beyond to help its fellow community members.

The Foundation is also committed to continuing to support the FreeBSD Project and community. In addition to publishing the *Journal*, we are increasing funding for development projects, working on new methods for online advocacy and continuing our company outreach. To find out more about these projects, please take a look at the Foundation's Q1 2020 Status Report and our March/April Newsletter.

We also want to thank the *Journal* authors, editorial board, and publishers for their hard work in getting this issue out during an unpredictable time.

Finally, I want to wish everyone good health for you and your loved ones. Thank you again for your ongoing support. We'll be here continuing to make big strides to improving FreeBSD and supporting the growing community.

We hope you enjoy the issue!

Deb Goodkin
Executive Director
FreeBSD Foundation

FreeBSD® JOURNAL
March/April 2020

15 OpenZFS Encryption Arrives on FreeBSD
The FreeBSD storage encryption ecosystem previously dominated by GELI now has a new contender. *By Allan Jude and Kyle Kneisl*

22 FUSE Driver Update
FUSE is a driver and a protocol for allowing userspace processes to implement a file system which the kernel presents to other processes just like any other. *By Alan Somers*

27 The Humble Bridge
`if_bridge` does not scale as well as it could. In fact, it currently scales very badly, indeed. *By Kristof Provost*

3 Foundation Letter
Thank You to the Community *By Deb Goodkin*

5 New Faces of FreeBSD
In this installment, the spotlight is on **Alfredo Dal'Ava Júnior**, who received his src bit in January, and **Ryan Moeller**, who received his src bit in February. *By Dru Lavigne*

8 FreeBSD Foundation Development Project Update: Toolchain Modernization
By Deb Goodkin

11 Conference Report: Scale 18x
By Deb Goodkin

12 We Get Letters
Dear Congratulations-on-Learning-to-Ask-A-Question *By Michael W Lucas*

33 Events Calendar
By Dru Lavigne and *Anne Dickison*

new faces of FreeBSD

BY DRU LAVIGNE

This column aims to shine a spotlight on contributors who recently received their commit bit and to introduce them to the FreeBSD community.

In this month's column, the spotlight is on Alfredo Dal'Ava Júnior, who received his src bit in January, and Ryan Moeller, who received his src bit in February.

Tell us a bit about yourself, your background, and your interests.

• Alfredo: I'm Alfredo Dal'Ava Júnior (adalava, alfredo@freebsd.org), born in Poços de Caldas / MG / Brazil and living the last 10 years in the metropolitan area of Campinas / SP / Brazil. I am married and have two children. I have developed software professionally since I was young and worked my way through many programming languages while making some small contributions to open source projects. My background is in Computer Science, but I have always liked the iteration between the machine and the real world, so I have stayed in close touch with electronics, sensors, actuators, and machines in general.

I am a curious person, and my curiosity has led me to good experiences in several areas. For a long time, I worked with networks, internet service providers, implementation of network protocols and services, automation, data storage systems, embedded systems. I also experimented with robots, drones, and machine learning.

Outside of work, I like musical instruments, mountain bikes, and more recently, monitoring meteors with video cameras and radio frequency, generating scientific data for study by some networks, and gradually deepening my understanding of these objects that enter the Earth's atmosphere every day.

• Ryan: I'm 31 years old, got my first computer when I was 10, and started to develop an interest in computer hardware and software shortly after that. I have always been interested in electronics and mechanics and puzzles, and that's basically what computers are made of. Through middle school and high school, I volunteered at community theaters as a backstage hand, doing set construction, rigging, light ing, and sound. I was hired as Master Electrician at the local playhouse and also did lighting design for their children's theater productions. After high school, I worked with several event production companies as a freelancer doing live video, lighting, and sound. I filled various roles including technical director, lighting designer, assistant director, sound designer, video editor, and graphic designer. For a while, I worked with a local DJ company doing live generative video projections. Eventually, the unstable income made it hard to continue as an independent contractor, so I decided to pursue a degree in Computer Science and a career in Software Development.

To the best of my recollection, my interest in development started when I was around the age of 12. My mom was learning HTML and I guess it started to rub off on me. It became a hobby. In middle school, I spent hours in the library/computer lab studying any books I could

find about programming (mostly old books on BASIC). I took the advanced computer classes and competed nationally in C++ programming competitions through the computer club in high school. After high school, I continued to develop software through hobby projects and occasional freelance web development projects. In my free time, I tinkered with game engines, audio processing, shader programs, embedded systems, operating systems, etc. When I finally went to finish up my CS degree, I had become aware of a gap in my knowledge where I was missing some of the fundamentals of data structures, algorithms, theory of computation, and math, and I was eager to fill it.

Now I am a software engineer on the OS/Services team at iXsystems. My main project has been the upstreaming of FreeBSD platform support into the OpenZFS (formerly ZFS on Linux) code base. I've particularly been focused on the build system, test suite, and automation. At this time of writing, tests are being passed and FreeBSD support is on the brink of being merged.

How did you first learn about FreeBSD and what about FreeBSD interested you?

• Alfredo: I was a heavy FreeBSD user in the early 2000s when I worked for a small software and automation company in my hometown that offered IT solutions to other companies in the city, including some ISPs. We had a subscription to a magazine that each month brought CDs with the latest versions of various open source operating systems and different distros. FreeBSD was one of those systems, and at some point, we decided to go with FreeBSD for some core network activities. Several attributes caught my attention such as the convenience of installing all at once a complete system with kernel sources and compiler ready to easily customize and optimize for that machine. At the time, we configured and installed dozens of machines to control network access, optimize and control bandwidth usage in the most remote and unfriendly locations, and to host websites and email servers.

• Ryan: I remember in high school I would bring a SLAX live CD to get around policies in the computer labs, such as to run Firefox instead of Internet Explorer and play games. My earliest recollection of FreeBSD though was after high school. I used to hang around the Insanely-Mac IRC and someone brought up using ZFS zvols for backing VMs. That led me to FreeBSD. I was thrilled by ZFS and by the ease of use of the network stack for doing silly things like bridges and such. I stumbled across the BSD Now podcast in my quest for knowledge and was hooked. My original storage zpool is still in use to this day, though it has been through a lot of trials and transformations.

How did you end up becoming a committer?

• Alfredo: The company I work for, Eldorado Research Institute, located in Brazil, has some partners that use FreeBSD and some coworkers had already contributed to the FreeBSD project. At the end of 2018, after many years away from FreeBSD, I was invited to collaborate on the efforts part-time. I started participating in IRC channels to understand how I could help and joined forces with other commiters like Brandon Bergren (bdragon), Justin Hibbits (jhibbits), Leandro Lupori (luporl), and Mark Linimon (linimon) on the POWER processor front.

For about one year, I contributed debugging problems, helped test hypotheses and patches, and created my own patches for different areas such as loader, kernel, base, ports, and LLVM, helping in whatever way I was able to make the transition from the old GCC4 to LLVM happen, always with the help of these and other committers. At one point, Justin Hibbits

kindly invited me to be mentored by him and I accepted the offer with great satisfaction and recognition of the work that was being done.

• Ryan: I was familiar with FreeBSD and had been using it and hacking on it and submitting patches here and there for years. In my senior year of college, I started as an intern at iX-systems on the OS/Services team and I joined the team full time a little under a year later. There, I was able to work on contributing more frequently to FreeBSD. Eventually, the crew at iX proposed me for a src commit bit. Now my colleagues mmacy@ and mav@ are my mentors! People joke about it being a punishment not an honor, but for me it was really a life goal achieved.

How has your experience been since joining the FreeBSD Project? Do you have any advice for readers who may be interested in also becoming a FreeBSD committer?

• Alfredo: Being active in an open source project is something new for me and I'm having a lot of fun making things work. FreeBSD has a team of competent developers who are willing to help, and that has greatly facilitated my learning to solve problems in the best way. If you have the opportunity of working in a company that allows you time to work in the project, that would be great!

For anyone who is interested in being a committer, I recommend checking the good documentation that is available, attending the IRC channels in the areas of interest, participating in discussions, subscribing and interacting on email lists, trying to fix bugs already reported, or helping to maintain an updated port. These are some ways to make yourself known and show your work. A commit bit is not required to contribute—developers are open to receive and review code sent by anyone, and when your code is ready and accepted, authorship is credited in the commit message. At some point, your commitment will certainly be recognized by a committer who will indicate your name and guide you through the process.

• Ryan: My experience has been great. The community has been warm and welcoming. I have had one frightening experience already: I am admittedly a subversion noob, and when it came time for me to make my first commit, my heart skipped several beats when I realized that unlike git there is no svn push after you commit, the commit happens globally. Fortunately, the commit was ready! So, my advice for newcomers: practice with subversion!

DRU LAVIGNE is the author of *BSD Hacks* and *The Best of FreeBSD Basics*.

Foundation REPORT

BY DEB GOODKIN

Development Project Update:

Toolchain Modernization

Removal of GCC 4.2.1 marks significant milestone in FreeBSD's move to an integrated, modern, permissively licensed tool chain.

As announced, on February 29, 2020 we removed GNU Compiler Collection (GCC) version 4.2.1 from the FreeBSD base system. Although GCC has not been used by default for anything in FreeBSD-current / svn head for some time before this commit, this milestone marks the completion of a major journey to modernize the FreeBSD toolchain.

The purpose of this short article is to provide detail about the motivation, scope, and timeline for this initiative. None of this is new, per se. Nevertheless, we felt it was important to pull all the relevant bits that are currently spread across mailing lists and man pages into one canonical piece for easy reference.

Motivation

License needs clearly played a role in motivating the Toolchain Project. Most readers will know that GCC adopted the strong copy-left GPLv3 license in 2007, presenting obvious incompatibilities with the ethos of this community.

A desire to provide a performant, modern toolchain played an important role as well. This is important to support research and development efforts and to give the project more freedom to drive the tool chain's roadmap. Many of the tools in the Toolchain project (see full list in the Scope section below) represent technical improvements. In most cases, they are relatively new projects without much cruft in their code bases, e.g. they only support operating systems, executable formats, and architectures people actually use today.

Take for example the new compiler, Clang/LLVM, which brings advantages like:

- Vastly improved warnings and error messages due to parser design compared to the pre-GPLv3 GCC version 4.2.1 that we were using. Contemporary GCC versions do not have these problems
- Compilation that is fast enough to use for syntax highlighting
- LLVM is always a cross compiler
- Ability to JIT code
- Blocks support

Extensible and modular design allowing developers and researchers the ability to integrate, reuse, or replace individual parts of LLVM

One industry observer goes so far as to credit the recent blossoming of new development languages like Rust, Kotlin, and Swift in part on LLVM. We are gratified that our design decision to get on board early with LLVM has positioned FreeBSD to be one of (if not *the*) best supported open source operating systems when using LLVM.

Foundation REPORT

Development Project Update:

Toolchain Modernization

Scope

In addition to Clang/LLVM, other tools in the Toolchain project are:

- The ELF Tool Chain Project – A project to create BSD-licensed replacements for GNU binutils, and that also adds some other tools
- libc++ – A BSD-licensed C++ standard library
- libcxxrt – A BSD implementation of the Itanium C++ ABI used by FreeBSD. This provides the low-level parts of the C++ implementation: exception handling, RTTI, etc.

This project also adds support for external toolchains, which provides significant benefits to embedded developers who often need to run vendor-provided toolchains. An explicit goal of external toolchain support is to support any modern compiler with a gcc compatible driver. Another goal is that using external toolchains should be easy.

Timeline

These efforts have been underway for some time, and so the February 29 announcement really marks the culmination of what has been years of effort by countless contributors.

Thanks to much hard work, FreeBSD has used Clang by default for x86 and LE arm targets since version 10.0, released in January 2014, and lld by default since 12.0.

Here's the complete timeline:

- May 2010 BSD Toolchain Summit and BSDCan and demonstration of ClangBSD
- Apr 2011, Build Clang by default on x86 and powerpc (Clang available, but FreeBSD still built with GCC)
- January 2012, FreeBSD 9 released with Clang available
- Feb 2012, WITH_CLANG_IS_CC option
- Nov 2012, Clang default for i386 and amd64
- Feb 2013, Build clang for LE arm
- Mar 2013, Clang default for arm
- Sep 2013, Don't build GCC when Clang is default
- Jan 2014, FreeBSD 10 released with Clang and without GCC on x86
- Jan 2015, Use a set of ELF Tool Chain tools by default
- Oct 2016, FreeBSD 11 released with ELF Tool Chain tools
- Nov 2016, Support for building and linking via LLVM IR
- May 2018, Enable lld as the system linker by default on i386
- Dec 2018, FreeBSD 12 released with lld as the linker on x86 and Arm
- Jan 2020, Clang and lld as the default toolchain for RISC-V
- Feb 2020, Retired GCC 4.2.1. All supported architectures either use in-tree Clang, or rely on external toolchain (i.e., a contemporary GCC version from ports)

Closing Thoughts

Clearly, license concerns were a big part of the initial motivation to replace GCC. But the FreeBSD Toolchain project has been and is about much more. Brooks Davis said it well:

"Comprehensive LLVM support in FreeBSD is enormously advantageous in the research world where LLVM is the research compiler of choice. In the DARPA SSITH program, three of five teams are using FreeBSD specifically due to our LLVM support. Further, our collaboration with Arm to bring CHERI to ARMv8-A in the form of the Morello prototype would have been much harder if not impossible without a permissively licensed operating system and toolchain."

Foundation REPORT

Development Project Update:

Toolchain Modernization

Finally, we would like to give a shout out to a few of the contributors without whose leadership and efforts, this major milestone would not have been possible.

FreeBSD Toolchain Hall of Fame

- Brooks Davis
- Roman Divacky
- Dimitry Andric
- Ed Maste
- David Chisnall
- Joseph Koshy
- Kai Wang

Honorable Mention

- Warner Losh

DEB GOODKIN is the Executive Director of the FreeBSD Foundation. She's thrilled to be in her 15th year at the Foundation and is proud of her hardworking and dedicated team. She spent over 20 years in the data storage industry in engineering development, technical sales, and technical marketing. When not working, you'll find her on her road or mountain bike, running, hiking with her dogs, skiing the slopes of Colorado, or reading a good book.

Thank you!

The FreesBSD Foundation would like to acknowledge the following companies for their continued support of the Project. Because of generous donations such as these we are able to continue moving the Project forward.

FreeBSD FOUNDATION

Are you a fan of FreeBSD? Help us give back to the Project and donate today! **freebsdfoundation.org/donate/**

Please check out the full list of generous community investors at freebsdfoundation.org/donors/

Uranium

intel

Iridium

NetApp

Platinum

NETFLIX

Gold

facebook JUNIPER NETWORKS

Silver

BECKHOFF Microsoft STORMSHIELD Tarsnap vmware

SCALE 18x Conference Recap

BY DEB GOODKIN

On March 4, members of the Foundation team and FreeBSD Community headed to Pasadena, California for a packed agenda of FreeBSD activities at SCALE 18x. We began the event by teaching an all-day *Getting Started with FreeBSD* workshop led by FreeBSD contributor Roller Angel, with FreeBSDSecurity Officer Gordon Tetlow , FreeBSD Foundation Founder and President Justin Gibbs, and myself. Despite current events, we had an excellent turnout and if you're looking for something to occupy your time right now, I've included the notes so you can do this at home.

That evening we ran a BoF called Why FreeBSD is For Me. We started at 8 p.m. and continued until the cleaning crew kicked us out. Over 15 people attended to learn more about our favorite open source operating system. To encourage attendee interaction, we formed a circle around all the tables and just let people jump in with their questions. We discussed some very thoughtful questions, such as What are our challenges? What SoCs work best with FreeBSD? Will the Raspberry 4 be supported? and many more.

We also had a FreeBSD table in the Expo Hall. Even with lower attendance, we enjoyed a lot of people stopping by our table. It didn't hurt that we had a big bottle of hand sanitizer that we shared with anyone who asked. Many students stopped by to tell us how much they enjoyed the workshop. Some wanted to get into tech. I met a math teacher who was interested in teaching some middle-school-level, hands-on workshops, so I'm excited to work with him and get him to one of the BSD conferences. I also spoke with a few people who were curious about how to contribute to the Project and quite a few who wanted to try out FreeBSD on their systems.

When I gave my talk *FreeBSD, The Other Unix-Like Operating System and Why You Should Get Involved* we unfortunately experienced a little technical difficulty with the projector not displaying my slides. Then, after what seemed like hours, I found a cable with a button on the podium and pushed it which magically made everything work!

In the end, SCALE was, yet again, another successful advocacy effort to encourage more people to use and contribute to FreeBSD.

As will unfortunately be the case for most of us in this industry now, SCALE was the last in-person event we will be able to attend for a while. Don't worry though, we're already working on how to make more online tutorials and how-to guides available so it will be easy for more folks to try out FreeBSD. In the meantime, please check out the how-to guides we already have!

DEB GOODKIN is the Executive Director of the FreeBSD Foundation. She's thrilled to be in her 15th year at the Foundation and is proud of her hardworking and dedicated team. She spent over 20 years in the data storage industry in engineering development, technical sales, and technical marketing. When not working, you'll find her on her road or mountain bike, running, hiking with her dogs, skiing the slopes of Colorado, or reading a good book.

WeGet letters

by Michael W Lucas

Dear Opinionated Doofus,

You've said a bunch of stuff against encrypting your disks. Are you crazy? With the world we live in, we absolutely need storage encryption.

—Encryption4Life

Dear Whatever,

The answer to the question you asked is obviously "yes."

Don't know what the other statements had to do with anything.

Dear Opinionated Doofus,

You wanna be that way? Fine. Do you honestly believe that encrypting disks is not worthwhile?.

—Encryption4Life

Dear Congratulations-on-Learning-to-Ask-a-Question,

This is two questions. The answer to the first is, again, yes.

As for the second:

The hard part of writing this column has nothing to do with putting the words together, explaining the technology, or dealing with our cutting-edge technology that's barely a sneeze from toppling over. No, it's pretending that I care what people think. Upon seeing my cheerful expression of incipient bucolicism, readers erroneously conclude that I'm amenable to their inadequately brewed opinions. In truth, I've learned that maintaining a gormless façade is all that protects and preserves my tolerable existence wedged in the Writing Pit against the League of Extraordinary Grumps declaring me their Grand Fiend by snarling acclaim.

If you think you need disk encryption everywhere, go for it. I won't argue with you. I can't be troubled.

Most at-rest disk encryption isn't useful for most people, though. And most environments where it would be useful don't use it.

Encryption is a response to a threat. Some of those threats are real. Some are not. Some of those threats are targeted at us. And everyone must balance different threats.

Some people have real, physical threats. Relief workers in war zones need encrypted storage. It won't save them, but might well save their coworkers, peers, and those they succor. Their computers are often offline or connect to the Internet only on occasions when the satellite uplink happens to be working and the ongoing atrocities have slowed to a trickle.

Folks with more connected lives face different threats. Disk encryption prevents some of them.

Disk encryption is great against random theft. If you carry confidential or sensitive documents on your laptop, disk encryption will keep a lucky mugger from uploading them to

WikiLeaks. Most muggers who discover that your laptop doesn't run a "normal" operating system will not consider themselves lucky. They will wipe the disk, saving you the worry.

Disk encryption is great for data you want to deliberately destroy. If your organization is bound by rules declaring decommissioned disks must be overwritten with garbage when removed from service, encrypting the disks at installation is proactive scrambling. Once you destroy the keys, the disks are unreadable.

For most people, key destruction is the problem.

Or rather, key loss.

People have this horrid habit of living at the outer limit of tolerable complexity. We keep claiming ownership of problems (usually branded as "opportunities") until we pick up one too many and complain that we are swamped, overwhelmed, and incapable of handling anything else. At that point, much like pushing two-gigabit through a one-gigabit link, we start dropping packets. Encryption key management is one of these packets. If you are going to use disk encryption, you must dedicate time and mental energy to managing and sustaining those keys.

I have known four people who legitimately manage their disk encryption keys and who regularly and demonstratably dedicate time, effort, and mental attention to the task. Three of them managed their disks under contract with their employment. The last was a successful union organizer who had been threatened and stalked by company owners.

I know many people who claim to manage their keys properly. Of those who make such claims, a substantial portion lose their decryption keys and their data. I've never performed a statistical analysis because I can't be bothered. Many recovered trivially because their data was not worthwhile. Others recovered trivially because the important chunks of data were backed up elsewhere.

Others lost critical data and never recovered it, because they allowed their lives to become overly complex and failed to maintain that encryption.

With a bunch of work, you could attract the attention of a nefarious, three-letter agency, a criminal cabal, or an organization rejected by Robert Ludlum as too ridiculous for his worst novel. Anyone reading this column has left fingerprints all over the Internet. You send all your traffic over onion routing? They'll skip identifying you by IP address and use personal information. On the darknet, nobody knows you're a Fed.

So, if you're under serious threat?

Take the threat seriously. Devote time, energy, and attention to it.

If your disk is fully encrypted, are your backups? How are those backups stored? Where are they? And who is pursuing you? Your data is only as secure as the least protected mechanism.

I'm all for privacy. I'm all for experimenting with disk encryption, discovering how much attention it demands, and learning if it's worthwhile for you. Discovery of data on your hard drive can threaten your privacy. Next to advertising networks and Internet-connected refrigerators, though, your hard drive is a trivial risk. You don't need to remember your secrets; any number of globe-spanning megacorps will do it for you!

How are the keys stored? On a can-opener flash drive you snagged at a random vendor's table at a slightly less random trade show? Forget the possibility that the flash drive contains malware. Is the drive reliable? I'll answer that for you: no, it's not. You need at least one backup key. You must regularly verify that the backup still works. That backup media is probably also as dubious as an Oracle salesman under quota the night before quarter end, so you need to be able to create new backup keys on media that will hopefully remain less defective for at least a day or two.

How are those backups protected?

Maybe you don't have a key on removable media. Perhaps your key is a passphrase. Only you know the passphrase. If you are seriously threatened, what will you do when a bunch of goons break out their Human Decryption Toolkit (a rubber hose, a pair of pliers, and an assortment of pointy bits they got off the free coupons at Harbor Freight)?

Maybe your threat comes from the sort of people who need warrants. Those people have learned how to seize your laptop while it's running.

Should you surrender your privacy and your data? No.

Should you protect it with disk encryption? Only if that's a real threat to your well-being. How do you know if it's a real threat? If you're willing to dedicate a slice of your precious complexity tolerance to maintaining that encryption, and actually carry out that maintenance, it's a real threat. Otherwise, it's a learning experience.

In truth: anything on a networked computer is not truly private.

No wonder the League of Extraordinary Grumps wants me.

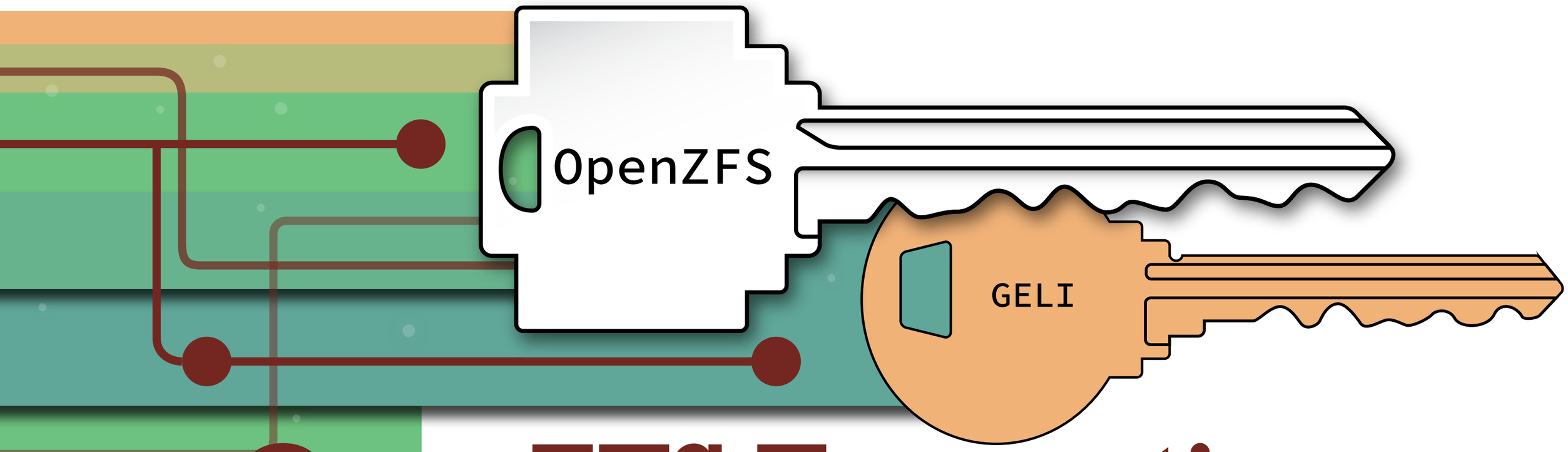
Have a question for Michael?
Send it to letters@freebsdjournal.org

MICHAEL W LUCAS (https://mwl.io)'s newest books are *Sudo Mastery, 2nd Edition* and *Terrapin Sky Tango*.

Write For Us!

Contact Jim Maurer with your article ideas.

(jmaurer@freebsdjournal.com)



OpenZFS Encryption Arrives on FreeBSD

BY ALLAN JUDE AND KYLE KNEISL

Among the new features arriving in OpenZFS on FreeBSD is a clever implementation of industry standard encryption at the dataset level. Effectively, this means that the FreeBSD storage encryption ecosystem previously dominated by GELI now has a new contender: OpenZFS "native encryption." In this article, we discuss how it compares to GELI from both a technical and usability perspective, equipping you to evaluate which of these encryption solutions you will prefer (spoiler: the new one, in most cases).

GELI

GELI has been FreeBSD's main native full disk encryption system for some time. GELI uses AES (typically AES-256 in either the XTS or CBC modes, which are particularly suited for encryption of disk data), paired with SHA-256 hashing (the same hashing that Bitcoin is based upon) and message authentication. To do this, GELI leverages FreeBSD's `crypto(9)` subsystem to provide hardware offloading, thereby greatly accelerating performance on systems that support AES-NI or otherwise utilize cryptographic acceleration (basically, any up-to-date platform).

However, GELI *has absolutely nothing to do with ZFS*. GELI encrypts an entire device, or at finer granularity, a partition on it. Thus, before first use of any filesystem protected in this way, the relevant devices or partitions have to be "unlocked." Once unlocked, the underlying filesystem is recognized and mounted, typically for the duration of the uptime. Unlocking procedures are governed by a master key and up to two user keys that variously involve keyfiles and passphrases. This complex set of keys allows for the possibility of re-keying a GELI partition (e.g., in the case of compromise of one key) without having to re-encrypt data.

Integrated in the FreeBSD bootstrap code, there is thus only a small `freebsd-boot` partition that remains unencrypted, containing just enough code to prompt for the GELI key passphrase, decrypt the boot partition, and read the bootloader or kernel from the underlying ZFS (or UFS) filesystem. This means that the entire filesystem (along with every byte of data on the relevant drives) is encrypted, except for the small chunk of startup code.

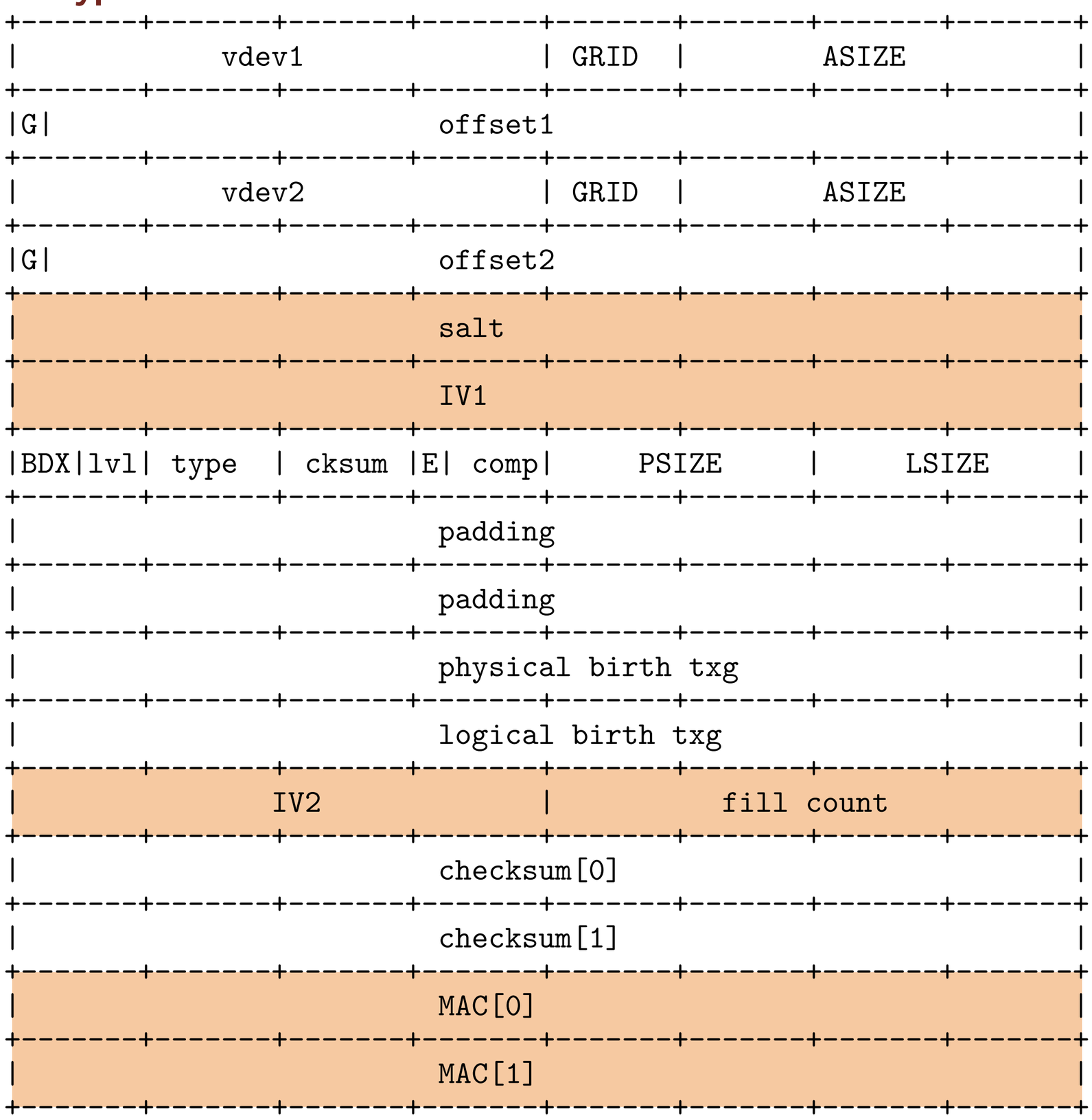
Properly implemented by the system administrator, GELI uses strong encryption, in strong modes, rotates data keys before they are overused, and has any other number of modern security practice elements in its implementation. Accordingly, GELI is, and will remain, a secure encryption implementation using best practices in all key areas.

Regular Block Pointer

```
+-------+-------+-------+-------+-------+-------+-------+-------+
|            vdev1              | GRID  |        ASIZE          |
+-------+-------+-------+-------+-------+-------+-------+-------+
|G|                      offset1                                 |
+-------+-------+-------+-------+-------+-------+-------+-------+
|            vdev2              | GRID  |        ASIZE          |
+-------+-------+-------+-------+-------+-------+-------+-------+
|G|                      offset2                                 |
+-------+-------+-------+-------+-------+-------+-------+-------+
|            vdev3              | GRID  |        ASIZE          |
+-------+-------+-------+-------+-------+-------+-------+-------+
|G|                      offset3                                 |
+-------+-------+-------+-------+-------+-------+-------+-------+
|BDX|lvl| type  | cksum |E| comp|     PSIZE     |     LSIZE     |
+-------+-------+-------+-------+-------+-------+-------+-------+
|                       padding                                 |
+-------+-------+-------+-------+-------+-------+-------+-------+
|                       padding                                 |
+-------+-------+-------+-------+-------+-------+-------+-------+
|                       physical birth txg                      |
+-------+-------+-------+-------+-------+-------+-------+-------+
|                       logical birth txg                       |
+-------+-------+-------+-------+-------+-------+-------+-------+
|                       fill count                              |
+-------+-------+-------+-------+-------+-------+-------+-------+
|                       checksum[0]                             |
+-------+-------+-------+-------+-------+-------+-------+-------+
|                       checksum[1]                             |
+-------+-------+-------+-------+-------+-------+-------+-------+
|                       checksum[2]                             |
+-------+-------+-------+-------+-------+-------+-------+-------+
|                       checksum[3]                             |
+-------+-------+-------+-------+-------+-------+-------+-------+

/*
 * Legend:
 *
 * vdev         virtual device ID
 * offset       offset into virtual device
 * LSIZE        logical size
 * PSIZE        physical size (after compression)
 * ASIZE        allocated size (including RAID-Z parity and padding)
 * GRID         RAID-Z layout information (reserved for future use)
 * cksum        checksum function
 * comp         compression function
 * G            gang block indicator
 * B            byteorder (endianness)
 * D            dedup
 * X            encryption
 * E            blkptr_t contains embedded data
 * lvl          level of indirection
 * type         DMU object type
 * phys birth   txg when dva[0] was written; zero if same as logical birth txg
 *              note that typically all the dva's would be written in this
 *              txg, but they could be different if they were moved by
 *              device removal.
 * log. birth   transaction group in which the block was logically born
 * fill count   number of non-zero blocks under this bp
 * checksum[4]  256-bit checksum of the data this bp describes
 */
```

Encrypted Block Pointer




```
/*
 * Legend:
 *
 * salt          Salt for generating encryption keys
 * IV1           First 64 bits of 96-bit encryption IV
 * X             Block requires encryption handling (set to 1)
 * E             blkptr_t contains embedded data (set to 0, see below)
 * fill count    number of non-zero blocks under this bp (truncated to 32 bits)
 * IV2           Last 32 bits of encryption IV
 * checksum[2]   128-bit checksum of the data this bp describes
 * MAC[2]        128-bit message authentication code for this data
 */
```

OpenZFS Native Encryption

By mid-2019, ZFS-on-Linux (ZoL) 0.8 introduced native encryption support; that is, encryption of ZFS datasets themselves on supported Linux platforms. Thanks to the efforts of a number of people from ZoL and FreeBSD, this important capability will soon be realized for FreeBSD in version 13, expected around the summer of 2020.

At the level of a filesystem encryption solution, we do not encrypt entire devices as in GELI. This is already immensely compelling; it is common to only desire encryption on a subset of sensitive data, and that data is probably naturally segregated into its own dataset as it is! Acting entirely within a dataset, native encryption is realized as what feels like just another ZFS dataset property. Amazingly, the fundamental layout of ZFS's data structures need not be expanded to accommodate encryption! Native encryption is accomplished through a clever re-

purposing of slack space existing in the structure of ZFS block pointers. In particular, those familiar with some of the less frequently used ZFS dataset properties have surely encountered the `copies` parameter, which by default is set to 1, but may also optionally be set to 2 (rarely) or 3 (almost never). As it happens, the space in the data structure pointing to the third copy (relevant only in the almost-never-used case of `copies=3`) is of the correct size to implement an AES-based encryption strategy. So, for the price of losing the rarely needed ability to store data blocks in triplicate (plus parity!), we gain filesystem encryption. Like other dataset properties, this property is inherited by child datasets. In fact, the only way in which the encryption property is notably different from other familiar ZFS dataset properties is that the use of encryption must (understandably) be set at creation time and cannot be unset. Accordingly, the ZFS data and metadata within the dataset are encrypted while at rest until mounted, but the existence of the dataset, and the ZFS properties for the dataset (e.g., the `logicalused` and other properties that might reveal the size and scope of data contained within) are necessarily always visible in the operating system.

One of the most game-changing features of ZFS native encryption is that, surprisingly, the dataset may be scrubbed or resilvered *while it is unmounted* and hence still encrypted. This unexpected feat is accomplished by splitting what was once a 256-bit hash field into a still more-than-ample 128-bit hash juxtaposed with a 128-bit message authentication code (MAC). This means that a ZFS administrator, unlike in the case of GELI, need not have access to client decryption keys in order to perform ZFS maintenance tasks—quite a compelling feature indeed. Furthermore, when sensitive data need not be accessed for a time, the dataset can be routinely unmounted where it subsequently becomes encrypted-at-rest, inaccessible to those without the key; this is not easily possible with GELI where typically an entire filesystem is decrypted at boot time and remains decrypted during the entire system uptime. Additionally, with GELI, the entire filesystem is protected by one set of keys, giving little granularity for access to different users; a user can either access the entire system, or none of it. With Native ZFS dataset encryption, different datasets can easily be given distinct keys. Those keys may then be provided to a given user in whatever access combination is appropriate.

At the same time ZFS native encryption is being introduced, ZFS `send` will now have the ability to send "raw" data blocks; that is to say, blocks are not interpreted in any way, and are sent as-is to the receiving system. In the case of natively encrypted datasets, this means that the key does not need to be loaded and data blocks can remain encrypted while in transit over the network, even when that transit (by accident or design) is itself not encrypted. This allows sensitive data to be backed up to a remote ZFS system without the remote system or its administrator (and, as has already been said, the local administrator) ever having access to the plaintext. This is a significant security posture improvement over GELI-based solutions.

(n.b.: the FreeBSD boot code does not yet support booting from an encrypted dataset, but it is likely that even that will be possible in the future!)

Getting Your Hands on OpenZFS 2.0

At the time of writing, it will likely be a few months before OpenZFS 2.0 is released and then integrated into FreeBSD's source tree. For those unwilling to wait, you can install the pre-release version of OpenZFS on FreeBSD 12.1 (and later) using the `sysutils/openzfs` package. If not using a `-RELEASE`, it is best to compile `sysutils/openzfs` and `sysutils/openzfs-kmod` from ports (because the module needs to exactly match the kernel it will run against). Once installed, replace `zfs_load="YES"` with `openzfs_load="YES"` in `/boot/loader.conf`. The port or package will install to the standard ports prefix, `/usr/local`, so the PATH environment variable may need adjustment to ensure the shell finds `/usr/local/sbin/zfs` before the sys-

tem-provided `/sbin/zfs`. Once a pool contains any encrypted dataset, the pool itself (including its unencrypted datasets) can no longer be imported on a system that does not support the encryption feature flag. However, as is often the case with new ZFS features, once the last dataset using the encryption feature is destroyed, the encryption feature flag will revert from "active" to "enabled" and the pool will once again be able to be imported by older versions of ZFS. Reminder: pools created with OpenZFS 2.0 will enable new features that may not be easily backward-compatible, and thus this type of experimentation should be done cautiously. The `zpool` man page describes how to selectively disable individual feature flags at pool creation time.

Transitioning a Dataset to OpenZFS Native Encryption

To transition data from an existing unencrypted dataset (remember: GELI-encrypted devices contain unencrypted datasets at the filesystem layer, so this applies to GELI pools too!), start by creating a new dataset with OpenZFS native encryption enabled:

```
zfs create -o encryption=aes-256-gcm -o keyformat=passphrase poolname/newdataset
```

Then, create a snapshot of the original dataset and use ZFS replication to copy its contents to the new encrypted dataset. Once complete, you are free to destroy the original dataset, rename the encrypted dataset to the original name, and update its mountpoint. Don't forget to consider the security ramifications of the resulting unallocated space which may be expected to contain unencrypted versions of your data; you will probably want to overwrite this space. The `zpool initialize` command is one reasonable way to accomplish this.

If you are transitioning from GELI, recall that GELI resides in a layer preceding the filesystem and thus does not interact with the filesystem. Accordingly, the only way to remove GELI is for each GELI-encrypted device (or partition) to rematerialize as an unencrypted device. Fortunately, ZFS itself can be our partner in accomplishing this. For example, a sufficiently redundant pool can strategically leverage a sequence of `geli kill` and `zpool replace` commands, removing the GELI layer from each device in turn. For pools with good levels of ZFS redundancy, this is relatively easy and can be safely done in-place. For pools with insufficient or no redundancy, the procedure will require (or be more safely performed with) a temporary, appropriately sized helper disk.

Conclusion

GELI is great and has been with us a long time. As it encrypts entire devices (including the operating systems on them in most cases), it is certainly the way to go, for example, when your aim is to protect every byte on a stolen (or confiscated) laptop or an RMA'd disk. However, it provides little granularity as a security solution, and there is considerable risk in having "encrypted" data in an accessible state for the entire uptime of a system. For example, a hacker with endpoint access to devices mounted under the GELI regime will in principle have full access to all data.

OpenZFS native encryption, on the other hand, gives the administrator more control over the granularity of what is encrypted and what combinations of users might access it. It also allows individual encrypted datasets to be unmounted when not in use, where they are protected "at rest," while the remainder of the system remains accessible. This is more sensible than GELI for file servers that hold sensitive data whose uptimes can extend into months or potentially years. A hacker with unfettered endpoint access (either physical or network) would see what an unprivileged user would see of the dataset when it is unmounted. The only disadvantage to using

OpenZFS native encryption (as it currently stands) is that it necessarily leaks certain information about the nature of the data it encrypts: that the dataset exists in the first place, the name of the dataset, the names of any snapshots, how much data there is, when it may have been created, how compressible it was, and so on. Typically speaking, this type of information tends to be of little consequence—the fact that a server houses, for example, health records of 4,000 patients, or surveillance videos, is of little sensitivity, but the actual contents of those records and videos would be. It is also very hard to argue against any encryption solution that requires fewer system administrators (as opposed to content owners) be in a position to access data.

It seems difficult to envision a use case (beyond the aforementioned stolen laptop) where GELI makes as much sense and is as usable and scalable as OpenZFS native encryption. While GELI will certainly remain with us as a trusted and reliable encryption solution for devices, the elegance of a ZFS-based encryption solution granular to individual datasets is extremely compelling and likely to become a fixture of ZFS user expectations moving forward.

ALLAN JUDE is VP of Engineering at Klara Inc., a global FreeBSD Professional Services and Support company. He also hosts the premier weekly BSD podcast, BSDNow.tv, and was elected to the FreeBSD Core team in 2016 and 2018. He is the co-author of *FreeBSD Mastery: ZFS* and *FreeBSD Mastery: Advanced ZFS* with Michael W Lucas.

KYLE "DRKK" KNEISL holds a PhD in mathematics and works as a scientist in the Washington, DC, area. He has been an active member and advocate of the FreeBSD and FreeNAS communities since 2013.

FreeBSD®

The FreeBSD Project is looking for

- Programmers
- Testers
- Researchers
- Tech writers
- Anyone who wants to get involved

Find out more by

Checking out our website
freebsd.org/projects/newbies.html

Downloading the Software
freebsd.org/where.html

We're a welcoming community looking for people like you to help continue developing this robust operating system. Join us!

Already involved?

Don't forget to check out the latest grant opportunities at
freebsdfoundation.org

Help Create the Future. Join the FreeBSD Project!

FreeBSD is internationally recognized as an innovative leader in providing a high-performance, secure, and stable operating system.

Not only is FreeBSD easy to install, but it runs a huge number of applications, offers powerful solutions, and cutting edge features. The best part? It's FREE of charge and comes with full source code.

Did you know that working with a mature, open source project is an excellent way to gain new skills, network with other professionals, and differentiate yourself in a competitive job market? Don't miss this opportunity to work with a diverse and committed community bringing about a better world powered by FreeBSD.

The FreeBSD Community is proudly supported by

FreeBSD FOUNDATION

Support FreeBSD®

Donate to the Foundation!

You already know that FreeBSD is an internationally recognized leader in providing a high-performance, secure, and stable operating system. It's because of you. Your donations have a direct impact on the Project.

Please consider making a gift to support FreeBSD for the coming year. It's only with your help that we can continue and increase our support to make FreeBSD the high-performance, secure, and reliable OS you know and love!

Your investment will help:

- Funding Projects to Advance FreeBSD
- Increasing Our FreeBSD Advocacy and Marketing Efforts
- Providing Additional Conference Resources and Travel Grants
- Continued Development of the FreeBSD Journal
- Protecting FreeBSD IP and Providing Legal Support to the Project
- Purchasing Hardware to Build and Improve FreeBSD Project Infrastructure

Making a donation is quick and easy.

freebsdfoundation.org/donate

FreeBSD™
FOUNDATION

FUSE Driver Update

BY ALAN SOMERS

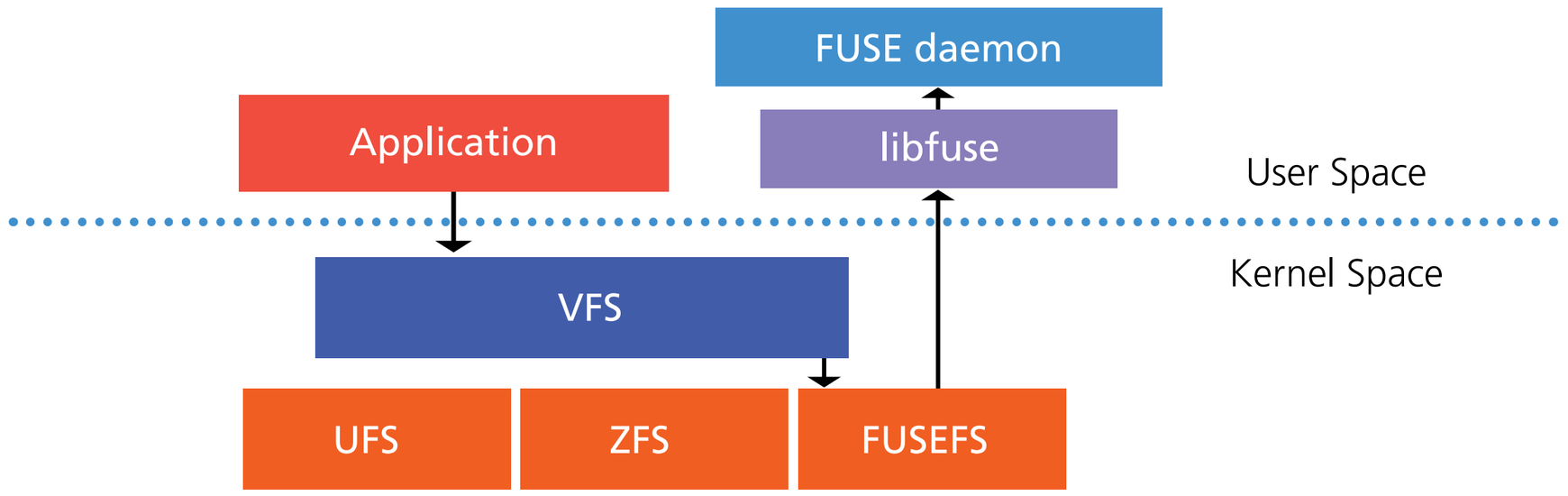
Have you ever mounted an NTFS-formatted hard drive on your FreeBSD or Linux laptop? Or have you ever connected your PTP camera to your laptop and browsed its photos through your file browser? Maybe you've used a distributed file system like MooseFS, GlusterFS, or CephFS? If you answered yes to any of those questions, then you've already used FUSE.

What Is FUSE?

FUSE (Filesystem in USErspace) is a driver and a protocol for allowing userspace processes to implement a file system which the kernel presents to other processes just like any other. Running in userspace makes file systems considerably easier to develop and debug than kernelspace file systems. That's why file systems like NTFS use FUSE. Userspace programs can also access libraries and utilities that aren't normally available in kernelspace, enabling virtual file systems like sshfs (which mounts a remote server's files over an SSH connection) and encfs (an encrypted file system that uses an ordinary file system as its backing store). Finally, FUSE provides license hygiene for kernel modules. For example, a GPLv3 kernel module cannot coexist alongside either a CDDL or a GPLv2 module; the result would not be legally redistributable. But a GPLv3 FUSE file system is not a derived work of the kernel, so it's perfectly redistributable.

FUSE was originally written for Linux in 2005, first appearing in kernel version 2.6.14. It proved so useful, however, that ports soon began to appear. Today, OSX, OpenBSD, Illumos, Minix, and of course FreeBSD also support FUSE. NetBSD does not use the FUSE protocol, but it still supports many FUSE file systems via a userspace compatibility layer. And a FUSE driver for DragonflyBSD is under development.

FUSE API Stack



FreeBSD's FUSE Port

FreeBSD's FUSE driver began life as a GSoC project in 2005 by Csaba Henk but wasn't integrated into the base system. A further GSoC project in 2011 by Ilya Putsikau finished the port and Attilio Rao merged it soon after. However, the 2011 version was still a few years behind the then-current protocol and had some unresolved bugs. In the subsequent 8 years, many

of those bugs went unaddressed and there was little maintenance or new features until now. Thanks to Foundation sponsorship, I was able to rectify this situation. I've largely rewritten the driver, updated the protocol version, fixed dozens of bugs, and added new features and performance enhancements.

Using FUSE File Systems

Using FUSE file systems is fantastically easy—that's the whole point of FUSE. Once mounted, they can be accessed just like a normal file system. The mount command varies between different file systems. For example, to mount an ext2 file system:

```
sudo pkg install -y fusefs-lkl e2fsprogs
truncate -s 1g /tmp/ext2.img
mkfs.ext2 /tmp/ext2.img
mkdir /tmp/mnt
lklfuse -o type=ext2 /tmp/ext2.img /tmp/mnt
```

Notice the lackof `sudo` in the second command. That's intentional. To enable this behavior, set the `vfs.usermount` sysctl to 1. FUSE daemons can run unprivileged. When they do, the mountpoint is only accessible by the user running the daemon. That's to prevent the daemon's user from spying on the I/O of other users. In fact, many FUSE daemons eschew any explicit permission checks in this mode, allowing the mounting user to do virtually whatever he wants with the file system, like this:

```
install -m 755 -o root -g wheel -d /tmp/mnt/bin
install -m 755 -o root -g wheel /bin/sh /tmp/mnt/bin/sh
```

The unprivileged user can create files owned by root! At first glance, that looks like a whopping security hole. But it's actually ok, since other users can't access that file system at all. An astute and paranoid reader might ask whether the mounting user can set create a SUID file and elevate his privileges that way. Rest assured—he can't. Unprivileged mounts automatically get `nosuid` set. All that's really happening is that the user is changing /tmp/ext2.img, which he owns. This feature is very cool. For example, you can use it to create a complete bootable image, such as for an embedded system.

Of course, that's only one use case. More traditional mounts are possible. For example, if a certain file system weren't available in the running kernel, you could mount a FUSE implementation as root with the `allow_other` and `default_permissions` options. That way it would be available to all users, and would function just like any other filesystem:

```
umount /tmp/mnt
sudo lklfuse -o type=ext2,allow_other,default_permissions \
/tmp/ext2.img /tmp/mnt
```

Developing FUSE File Systems

Compared to an in-kernel file system, developing file systems for FUSE is much easier. And not only is it easier to write a FUSE file system, it's also very easy to write it portably. Most FUSE file systems need few to no changes in order to run on several operating systems.

One of the benefits of programming in userland is the program is not limited to C. Indeed,

FUSE bindings are available for Perl, Python, Rust, Javascript, Java, Ruby, Nim, C#, Go, and probably others too.

Starting a FUSE daemon is slightly complicated: the daemon first opens `/dev/fuse`, then calls `nmount` with that file descriptor as one of the arguments. Then it begins to read FUSE requests from that same file descriptor and write the responses back. However, developers rarely need to worry about those details, because libfuse takes care of it. Instead, whether writing in C or another language, the developer generally just has to define callbacks for each supported FUSE operation. Then, the library takes care of all the plumbing. For example, in Python the crucial code for a "Hello World" example is just 37 lines. (For the full example, see https://github.com/libfuse/python-fuse/blob/master/example/hello.py).

```
class HelloFS(Fuse):

    def getattr(self, path):
        st = MyStat()
        if path == '/':
            st.st_mode = stat.S_IFDIR | 0o755
            st.st_nlink = 2
        elif path == hello_path:
            st.st_mode = stat.S_IFREG | 0o444
            st.st_nlink = 1
            st.st_size = len(hello_str)
        else:
            return -errno.ENOENT
        return st

    def readdir(self, path, offset):
        for r in  '.', '..', hello_path[1:]:
            yield fuse.Direntry(r)

    def open(self, path, flags):
        if path != hello_path:
            return -errno.ENOENT
        accmode = os.O_RDONLY | os.O_WRONLY | os.O_RDWR
        if (flags & accmode) != os.O_RDONLY:
            return -errno.EACCES

    def read(self, path, size, offset):
        if path != hello_path:
            return -errno.ENOENT
        slen = len(hello_str)
        if offset < slen:
            if offset + size > slen:
                size = slen - offset
            buf = hello_str[offset:offset+size]
        else:
            buf = b''
        return buf
```

The security model of FUSE may come as a surprise: by default, the daemon is responsible for authorizing all operations. That does place an extra burden on the FUSE file system developer. However, the usual behavior can be achieved by always using the `default_permissions` mount option. The upside is that the FUSE file system can use exotic authorization strategies, like bespoke ACL formats. This feature is also very useful for networked file systems that do authorization on the server, rather than on the clients.

New Features

FreeBSD 13's new fusefs(5) driver adds several new features, all of which should be immediately useable by existing FUSE file systems. Here are the most interesting:

- Kernel-side permissions checks (`-o default_permissions`) is now fully implemented.
- `mknod(2)`, `pipe(2)`, and `socket(2)` are now supported, so it's possible to create any type of file on a fusefs file system.
- Server-side support for `fcntl(2)` advisory locks has been added. Previously, fcntl locks were always implemented in-kernel, but that was insufficient for network file systems that do distributed locking.
- When mounted with `-o intr`, and if the server supports it, fusefs mounts are now fully interruptible. That means that a signal can interrupt an operation that's blocked waiting for a response from the server. It's similar to NFS's mount option of the same name.
- A fusefs mountpoint can now be exported over NFS.
- The kernel will now cache file names and attributes if the server allows it. The server can also asynchronously evict part of the kernel's cache. The kernel can also cache reads and writes, if permitted by the server. Finally, it will read ahead when a process appears to be reading sequentially. These features will all improve performance with no application or configuration changes required.

ALAN SOMERS has been a FreeBSD committer since 2013. In 2019, he rewrote the fusefs driver under contract from the FreeBSD Foundation. Currently Alan works for Axcient on their FreeBSD storage servers.

FreeBSD® JOURNAL

The FreeBSD Journal is Now Free!

Yep, that's right Free.

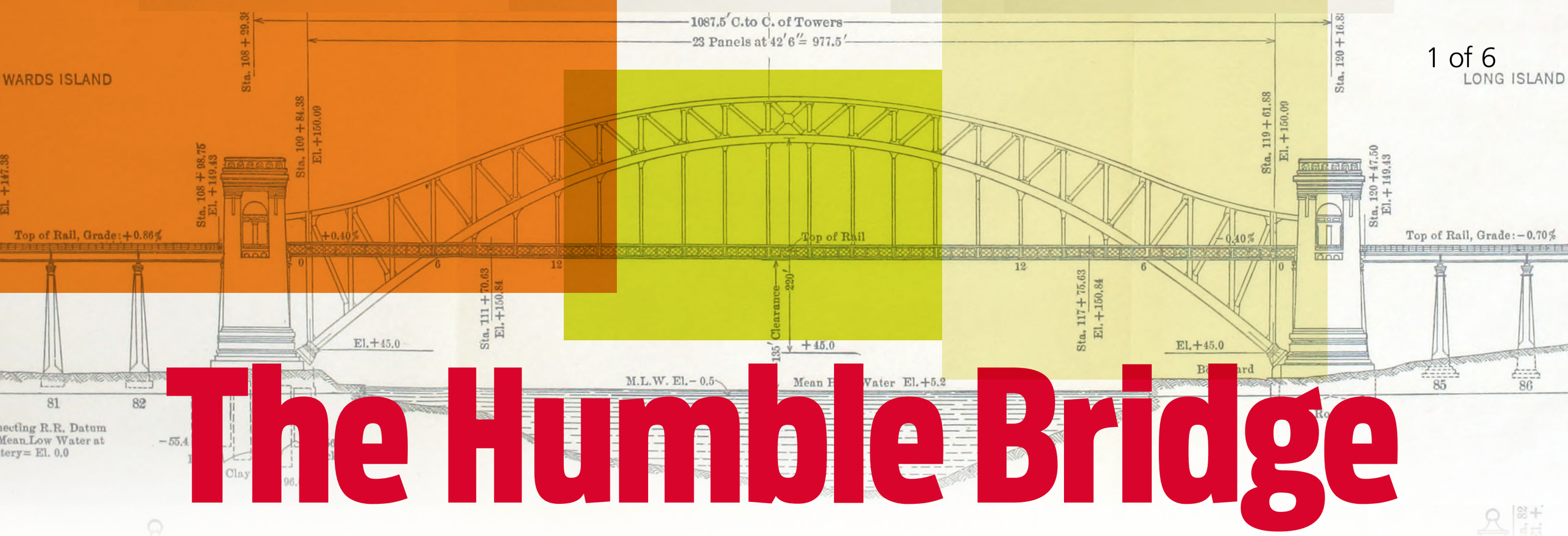
The voice of the FreeBSD Community and the BEST way to keep up with the latest releases and new developments in FreeBSD is now openly available to everyone.

DON'T MISS A SINGLE ISSUE!

2019 Editorial Calendar:

- Getting Started with FreeBSD (Jan/Feb 2019)
- Debugging and Testing (March/April 2019)
- FreeBSD for Makers (May/June 2019)
- Containerization (July/Aug 2019)
- Security (Sept/Oct 2019)
- Network Virtualization (Nov/Dec 2019)

Find out more at: freebsdfoundation/journal



The Humble Bridge

BY KRISTOF PROVOST

The humble bridge. Connecting places across rivers, ravines, chasms, valleys, ...

No, wait, we're not talking about that sort of bridge. if_bridge links together two or more Ethernet links (IEEE 802 networks, if you're that kind of person), in effect, turning your computer into a switch without all the messy cables.

One of the common use cases for if_bridge is to connect multiple virtual machines (or VNET jails) to a real network interface.

For example, this particular machine has a few bhyve VMs on it:

```
vmbridge: flags=8843<UP,BROADCAST,RUNNING,SIMPLEX,MULTICAST> metric 0 mtu 1500
    ether 02:ec:9a:db:86:01
    inet 172.16.1.1 netmask 0xffffff00 broadcast 172.16.1.255
    id 00:00:00:00:00:00 priority 32768 hellotime 2 fwddelay 15
    maxage 20 holdcnt 6 proto rstp maxaddr 2000 timeout 1200
    root id 00:00:00:00:00:00 priority 32768 ifcost 0 port 0
    member: tap5 flags=143<LEARNING,DISCOVER,AUTOEDGE,AUTOPTP>
            ifmaxaddr 0 port 11 priority 128 path cost 2000000
    member: tap4 flags=143<LEARNING,DISCOVER,AUTOEDGE,AUTOPTP>
            ifmaxaddr 0 port 10 priority 128 path cost 2000000
    member: tap2 flags=143<LEARNING,DISCOVER,AUTOEDGE,AUTOPTP>
            ifmaxaddr 0 port 7 priority 128 path cost 2000000
    groups: bridge
    nd6 options=9<PERFORMNUD,IFDISABLED>
```

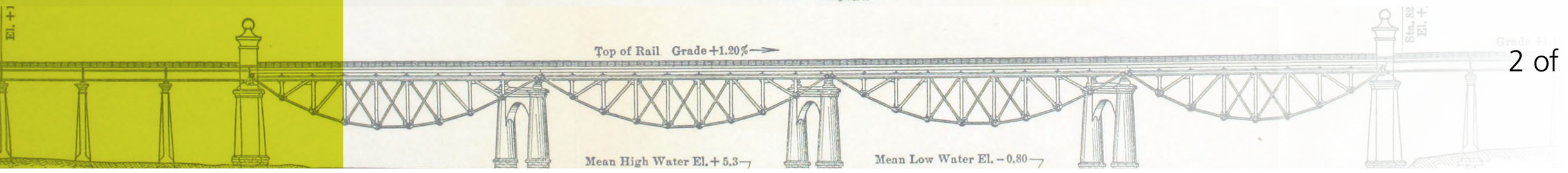
These VMs are connected to the tap2, tap4 and tap5 interfaces. They can all talk to each other on that particular (virtual) LAN, or to the host via the 172.16.1.1 address. The host machine also helpfully runs a DHCP server and a NAT firewall to translate traffic from the VMs to the public interface, allowing them to access the internet.

Problems

There's nothing particularly wrong with this setup, but if we didn't find a problem, this would be a very short and boring article.

Happily, there is a problem, or there can be a problem for some users. It's not visible, but if_bridge does not scale as well as it could. In fact, it currently scales very badly, indeed.

To demonstrate the problem, we'll create a simple test setup consisting of two machines connected with two Chelsio T62100 40Gbps links. The first machine will generate traffic on its



first interface and expect to receive it on the second. The second machine will use if_bridge to link its two interfaces.

The Chelseo driver supports netmap(4), which means we can use pkt-get (pkt-gen-g2017.08.06_1) to generate traffic. Netmap(4) is fast. I'm not going to prove that--you're just going to have to trust me. Yes, really. If you're going to doubt everything I say, we're never going to get anywhere!

So, with the following two commands (the first to receive traffic, the second to generate it) we can run a simple performance test:

```
./pkt-gen -i vcc1 -f rx -w 4 -W
./pkt-gen -f tx -i vcc0 -l 60 -d 192.19.10.1:2000-192.19.10.10 -s 10.10.10.1:2000-10.10.19.143 -S 01:02:03:04:05:06 -D 06:05:04:03:02:01 -w 4
```

We generate 60-byte packets. There's a good reason for this, and it's not just to be difficult.

Well, okay, actually it is precisely to be difficult. In general, it's easier for machines to transmit fewer large packets than many small packets. That's easy to understand, because each packet has a certain amount of overhead (e.g. context switches, working out where to send it, the all-important inter-packet nap, ...). So larger packets are more efficient. In your real network you want large packets, but when we're testing things, we're interested in the worst-case behavior because that's where we'll see the most impact of our attempts to improve things.

Once we run the test, we'll see output like this:

```
957.106303 main_thread [2666] 27.015 Mpps (28.730 Mpkts 12.967 Gbps in 1063499 usec) 497.42 avg_batch 99999 min_space
957.306302 main_thread [2666] 3.668 Mpps (3.901 Mpkts 1.761 Gbps in 1063498 usec) 24.62 avg_batch 949 min_space
958.169302 main_thread [2666] 27.016 Mpps (28.718 Mpkts 12.967 Gbps in 1062999 usec) 497.42 avg_batch 99999 min_space
958.369303 main_thread [2666] 3.669 Mpps (3.900 Mpkts 1.761 Gbps in 1063001 usec) 24.68 avg_batch 885 min_space
959.232304 main_thread [2666] 27.015 Mpps (28.717 Mpkts 12.967 Gbps in 1063002 usec) 497.37 avg_batch 99999 min_space
959.377805 main_thread [2666] 3.669 Mpps (3.700 Mpkts 1.761 Gbps in 1008502 usec) 24.66 avg_batch 950 min_space
960.280311 main_thread [2666] 27.011 Mpps (28.308 Mpkts 12.965 Gbps in 1048007 usec) 497.44 avg_batch 99999 min_space
960.441302 main_thread [2666] 3.668 Mpps (3.901 Mpkts 1.761 Gbps in 1063496 usec) 24.66 avg_batch 921 min_space
...
```

We're actually seeing output from both of our pkt-gen processes. One from the transmitting process, one from the receiving process. We manage to transmit 27 million packets per second (which adds up to nearly 13 gigabits per second). We only receive 3.7 million though. The remaining packets are lost.

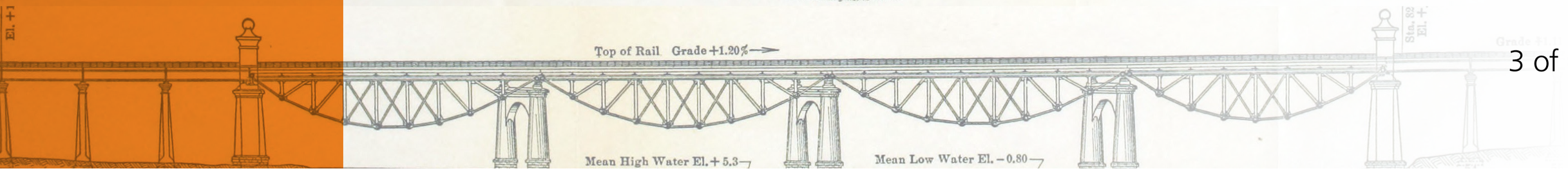
After 3.7 million brief moments of silence for those poor lost packets, we can ask why. If you remember only one thing from this article, remember that the most important question to ask when benchmarking is "Why?" "Why do we see that number?" "Why isn't it higher?" "Why isn't it lower?"

I happen to know, and you're just going to have to trust me, that if we'd route that traffic, we'd pass many more packets. That seems strange, because routing is harder work than switching.

Happily, FreeBSD has excellent tools that let us figure out where the system is spending its time. For this sort of work the `pmcstat(8)` tool is invaluable.

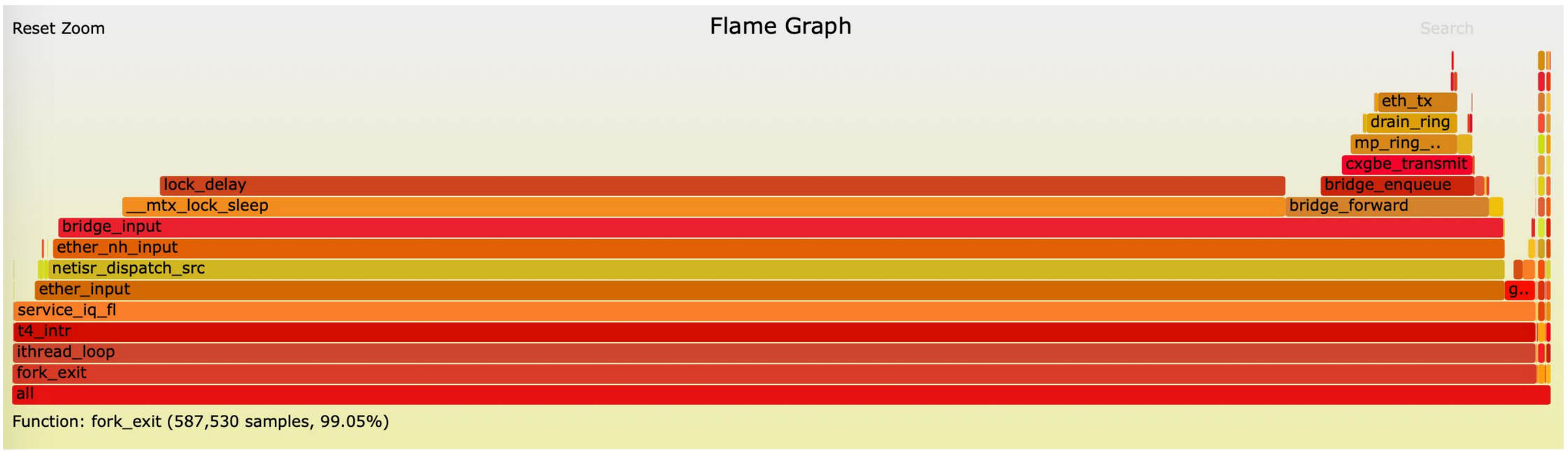
The following will produce a very instructive flame graph:

```
pmcstat -S cpu_clk_unhalted.ref_tsc -l 30 -z 50 -O data.out -q
pmcstat -R data.out -G data.stacks
stackcollapse-pmc.pl data.stacks > data.folded
flamegraph.pl data.folded > data.svg
```



The pmcstat tool is part of the base FreeBSD system. The stackcollapse-pmc and flamegraph tools are the work of Brendan Gregg (https://github.com/brendangregg/FlameGraph).

Anyway, pretty picture time:



This might look a little strange, but the most important thing it shows is where the system has been spending its time. The vertical axis shows a stack trace. Reading from the bottom it tells us that `fork_exit()` called `ithread_loop()` which called ...

Really, the bottom of the call stack isn't very interesting.

Let's work out what the system is doing when it could be passing around packets. Let's look for a function that seems to be taking a lot of time. The widest function--and thus the one where we spent most of our time—is `bridge_input()`. We spend 93% of our time there. That seems like it's related to bridging work. Why do we spend so much time in it? Most of the time we spend in `bridge_input()` we're actually doing `__mtx_lock_sleep()` rather than something useful-looking like `bridge_forward()`.

We shouldn't be napping. Why are we sleeping? And what's a 'mtx'?

Mutexes

`__mtx_lock_sleep()` is part of the mutex(9) code base. It's spelled 'mtx' in the function name because programmers are lazy. Mutexes provide mutual exclusion. They help ensure that only one CPU core can access a particular bit of memory at a time. Why is that required?

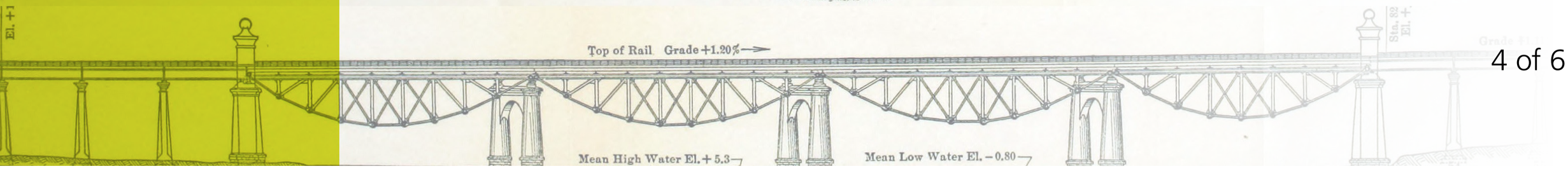
The simplest example is this: Imagine two CPU cores adding one to a number at the same time. Also imagine a universe where things always go wrong, where the toast always lands buttered side down. It may take some effort to imagine. No? You're there already? Fantastic.

So, we could have this order of operations:

```
CPU1: Load number (a = 1) from memory to a register (r0 = 1)
CPU2: Load number (a = 1) from memory to a register (r1 = 1)
CPU1: Increment register (r0 = 2)
CPU1: Store register (r0 = 2) to memory (a = 2)
CPU2: Increment register (r1 = 2)
CPU2: Store register (r1 = 2) to memory (a = 2)
```

Whoops. Suddenly 1 + 1 + 1 is 2. One way of resolving this problem is to ensure that CPU2 can't run this code at the same time, and that's what mutexes help us accomplish.

```
CPU1: Take mutex
CPU2: Try to take mutex. Fail and wait
CPU1: Load number (a = 1) from memory to a register (r0 = 1)
```




```
CPU1: Increment register (r0 = 2)
CPU2: Try to take mutex. Fail and wait
CPU1: Store register (r0 = 2) to memory (a = 2)
CPU1: Release mutex
CPU2: Take mutex
CPU2: Load number (a = 2) from memory to a register (r1 = 2)
CPU2: Increment register (r1 = 3)
CPU2: Store register (r1 = 3) to memory (a = 3)
CPU2: Release mutex
```

There are rather more complex flows in the if_bridge(4) code, but this gives an introduction to what mutexes are for.

The problem with mutexes is that while they ensure that the internal data structures can't be accessed by more than one CPU core at a time, that also means we can't do useful work on more than one core at a time.

It turns out that `if_bridge(4)` protects all of its internal data structures (such as 'this bridge connects `interface igb0` and `igb1`') with a single mutex, so no matter how many CPU cores you have only one of them can forward packets over the bridge at a time.

As a result, we can only forward about 3.9 million packets per second.

Improving This

One possible way of improving this is to change the mutex into a read/write lock (see `rw-lock(9)`). Read/write locks make a distinction between read accesses and write accesses. If code is only going to read the data and not modify it, we can allow multiple CPU cores to do so at the same time. It's only when data will be modified that we need to deny access to all other CPU cores, even the ones that only want to read the data.

That would already improve matters greatly. A prototype along those lines improved throughput to about 8 million packets per second.

Doing Better

FreeBSD 13 (which is not released yet, so is now CURRENT) comes with a new way to handle this sort of problem. It's called epoch(9), and was designed and implemented by people much, much smarter than I am.

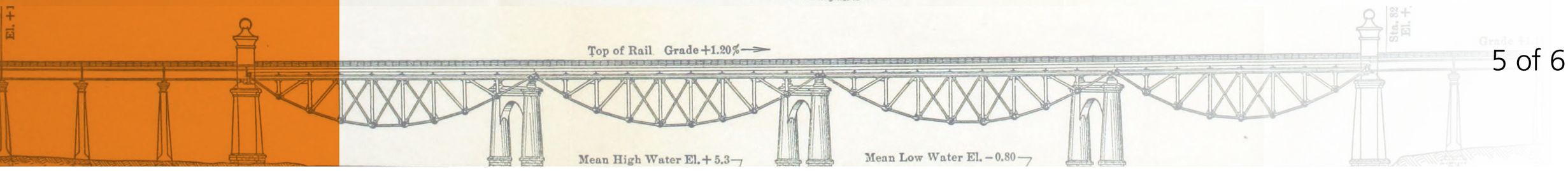
Let's pretend, for the purposes of this article, that I actually understand it, so I can explain what it does.

The epoch(9) facility originates from ConcurrencyKit (http://concurrencykit.org). When combined with the ConcurrencyKit lists (ck_queue), we can safely use protected data structures without acquiring a lock (either a mutex or a read/write lock) at all.

For if_bridge(4), the main data structures are lists, and the typical changes are to add or remove an item to or from the list.

If we add something to the list, we need to ensure that the list itself is always in a valid state, but we don't actually care too much if a given CPU core sees the item in the list or not. This means that as long as we're careful to fully populate new data structures before we insert them in the list, we can safely insert them while other CPU cores are looking through the list.

When we need to remove something from the list, we do need to be more careful. We can remove the data from the list, but we can't actually delete it until we're sure no one is using it anymore.



That's what the epoch(9) facility is for. When we want to read the protected data, we inform the epoch(9) system through the `NET_EPOCH_ENTER()` wrapper, that we're going to be accessing this data. When we're done, we also let it know, through `NET_EPOCH_EXIT()`.

This allows the system to keep track of who could possibly still be accessing the data we're going to remove. Very simply put, what it does it keep track of when we tell it we want to delete something (by registering a callback to do the final cleanup, using `NET_EPOCH_CALL()` it waits until every CPU core that called `NET_EPOCH_ENTER()` has also called `NET_EPOCH_EXIT()`. At that point we can be sure no one is using the data anymore, and we can safely delete it.

There are some additional complications because the ConcurrencyKit lists are safe to read from while they're being modified, but we cannot have multiple CPU cores attempt to change them at the same time.

In if_bridge(4), we use the existing mutex to protect write accesses to these lists. The final result is that we can still only perform one modification of the bridge state (e.g. adding an interface or learning which interface to use for a given MAC address) at a time, but we can keep processing packets on other CPU cores while we do this.

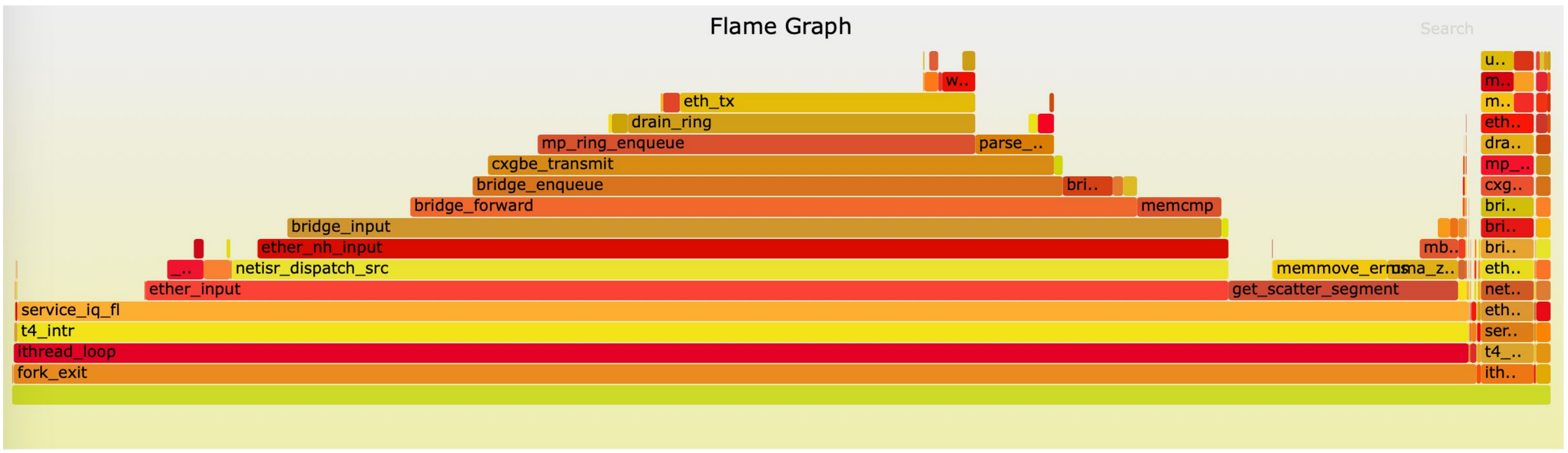
Remember that a single CPU core managed to process 3.7 million packets per second, so we can safely assume we can also handle many hundreds of thousands, or even millions, of changes to that state per second.

Final Measurements

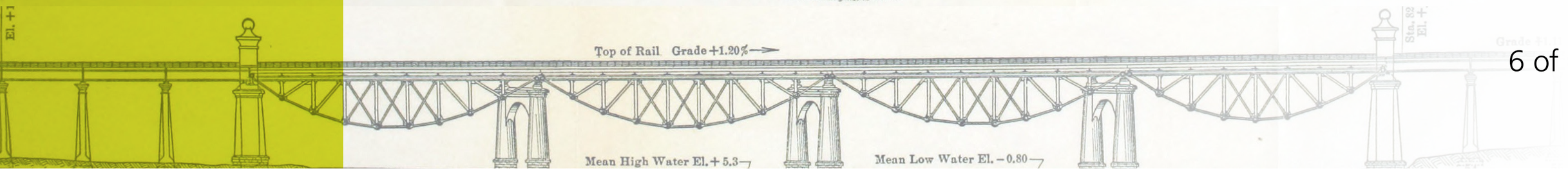
With the epoch-based approach in place the performance test shows:

```
283.745634 main_thread [2666] 18.637 Mpps (18.673 Mpkts 8.946 Gbps in 1001923 usec) 82.70 avg_batch 800 min_space
284.108710 main_thread [2666] 26.982 Mpps (27.009 Mpkts 12.951 Gbps in 1001000 usec) 494.37 avg_batch 99999 min_space
284.746709 main_thread [2666] 18.620 Mpps (18.640 Mpkts 8.938 Gbps in 1001076 usec) 82.80 avg_batch 816 min_space
285.172211 main_thread [2666] 26.981 Mpps (28.695 Mpkts 12.951 Gbps in 1063501 usec) 494.47 avg_batch 99999 min_space
285.747709 main_thread [2666] 18.640 Mpps (18.659 Mpkts 8.947 Gbps in 1001000 usec) 82.84 avg_batch 805 min_space
286.235212 main_thread [2666] 26.976 Mpps (28.676 Mpkts 12.949 Gbps in 1063001 usec) 494.34 avg_batch 99999 min_space
286.748709 main_thread [2666] 18.638 Mpps (18.656 Mpkts 8.946 Gbps in 1001000 usec) 83.01 avg_batch 800 min_space
287.236712 main_thread [2666] 26.976 Mpps (27.017 Mpkts 12.949 Gbps in 1001500 usec) 494.41 avg_batch 99999 min_space
287.750709 main_thread [2666] 18.633 Mpps (18.671 Mpkts 8.944 Gbps in 1002000 usec) 83.05 avg_batch 803 min_space
288.300211 main_thread [2666] 26.977 Mpps (28.690 Mpkts 12.949 Gbps in 1063499 usec) 494.41 avg_batch 99999 min_space
...
```

So, we now forward about 18.6 million packets per second, or a 5x improvement. There's also an associated pretty picture (or flame graph):



Here we see that we're no longer spending all of our time (or indeed any of our time) in __mtx_lock_sleep(), but instead we're doing useful work. Almost all of our time is spent doing work directly related to working out where the packets need to go and sending them there.



Closing

This article discusses ongoing work, so if you find a bug: well done! Tell the author as he might not have spotted it yet.

For those of you interested, the pending patches can be found here:

- https://reviews.freebsd.org/D24249
- https://reviews.freebsd.org/D24250

This project was made possible by sponsorship from the FreeBSD Foundation.

KRISTOF PROVOST is a freelance embedded software engineer specializing in network and video applications. He's a FreeBSD committer, maintainer of the pf firewall in FreeBSD and a board member of the EuroBSDCon foundation. He has an unfortunate tendency to stumble into uClibc bugs and a burning hatred for FTP. Do not talk to him about IPv6 fragmentation.

Jails ARE FBSD'S MOST LEGENDARY FEATURE:

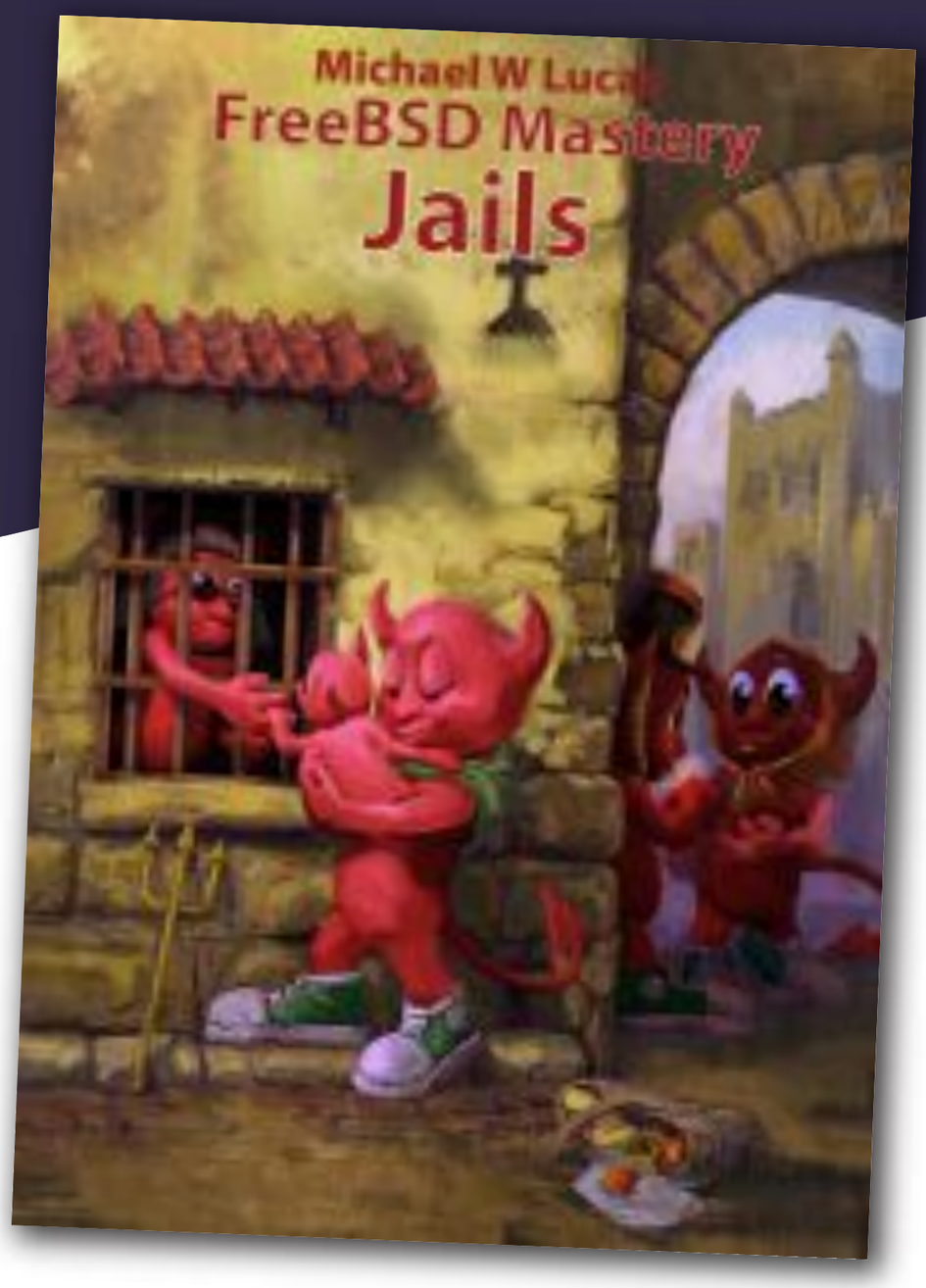
KNOWN TO BE POWERFUL, TRICKY TO MASTER, AND CLOAKED IN DECADES OF DUBIOUS LORE.

FreeBSD Mastery: Jails cuts through the clutter to expose the inner mechanisms of jails and unleash their power in your service.

Confine Your Software!

- Understand how jails achieve lightweight virtualization
- Understand the base system's jail tools and the iocage toolkit
- Optimally configure hardware
- Manage jails from the host and from within the jail
- Optimize disk space usage to support thousands of jails
- Comfortably work within the limits of jails
- Implement fine-grained control of jail features
- Build virtual networks
- Deploy hierarchical jails
- Constrain jail resource usage.

...And much, much more!



FreeBSD Mastery: Jails BY MICHAEL W LUCAS ***Available at All Bookstores***

2020 Events Calendar

BSD Events taking place through September 2020

BY ANNE DICKISON AND DRU LAVIGNE

Due to COVID-19, several annual conferences have been canceled or moved to a virtual format. Refer to each conference's website for the latest information. The Events page of the FreeBSD website contains information about new online events, such as the FreeBSD Office Hours.

BSDCan • June 3-6 • Virtual

https://www.bsdcan.org/2020/

This year, the 17th annual BSDCan will take place virtually. BSDCan hosts talks and tutorials on a range of topics based around the BSD family of operating systems.

Open Source Summit North America • June 29 -July 2 • Virtual

https://events.linuxfoundation.org/open-source-summit-north-america/

Open Source Summit is the leading conference for developers, architects and other technologists—as well as open source community and industry leaders—to collaborate, share information, learn about the latest technologies and gain a competitive advantage by using innovative open solutions.

EuroBSDCon • September 17-20 • Virtual

https://2020.eurobsdcon.org/

EuroBSDcon is the European annual technical conference gathering users and developers working on and with 4.4BSD (Berkeley Software Distribution) based operating systems family and related projects.